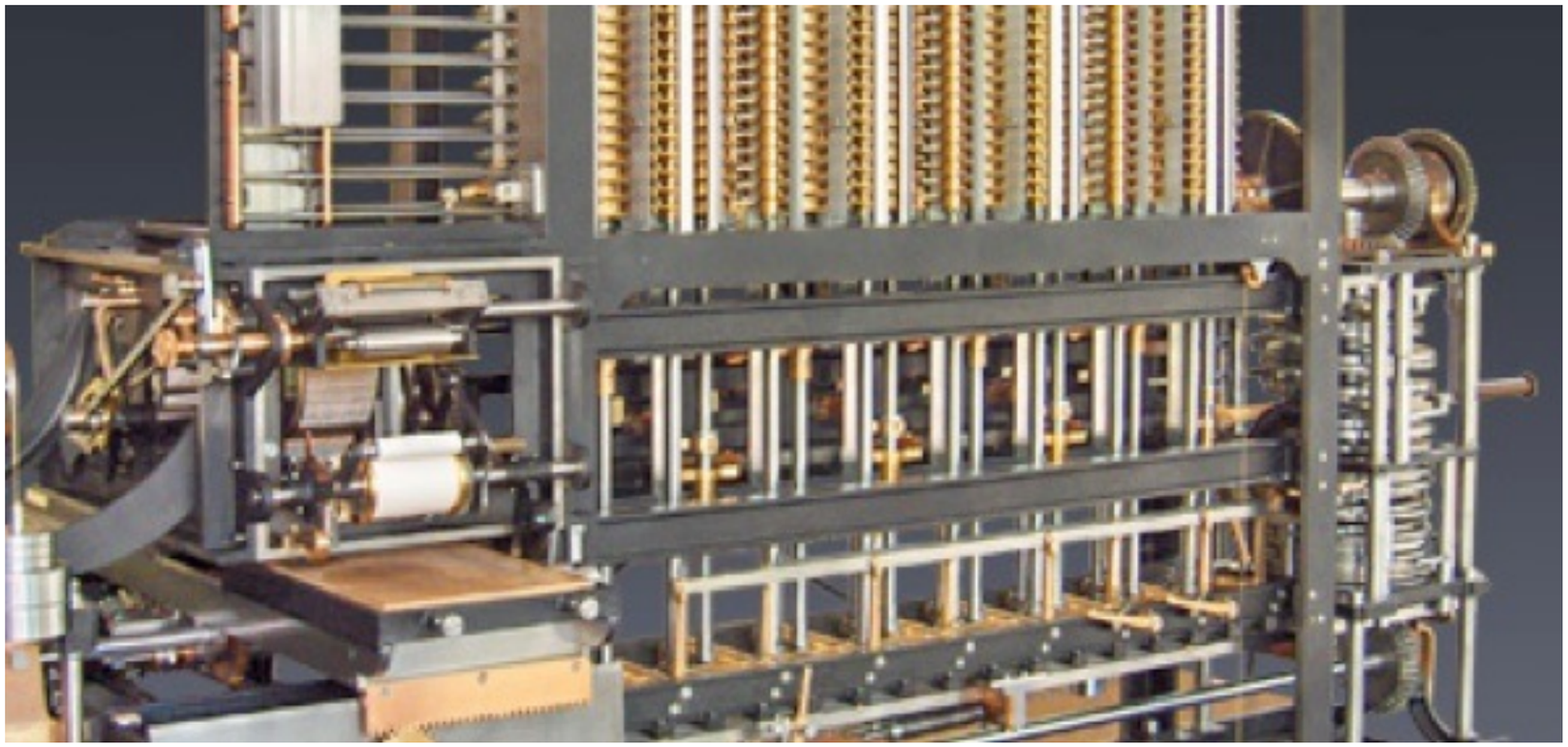
# UE Programmation C++

## Master Sciences pour l'Ingénieur






1er cours

Alice Cohen-Hadria alice.cohenhadria@gmail.com

Transparents de B. Gas

# UE Programmation C++

Master Sciences pour l'Ingénieur

## Description de l'UE



C++:

- 3 séances de cours (6h)
- 3 séances de TD (6h)
- 4 séances de TP (16h)
- Examens: 1 ctrl C++ & 1 Ctrl C++ Avancé. Une de note de TP

## Plan du cours



- Introduction, code compilé/interprété, fichiers (h, cpp)
- Placement en mémoire (global, local, dynamique)
- Notion de classe, instance, attribut, méthode et encapsulation
- Pointeurs, références et gestion dynamique
- Constructeur, destructeur, méthodes de classe
- Accès aux attributs
- Flux (ostream, cout, cin, ...)
- Héritage simple et polymorphisme

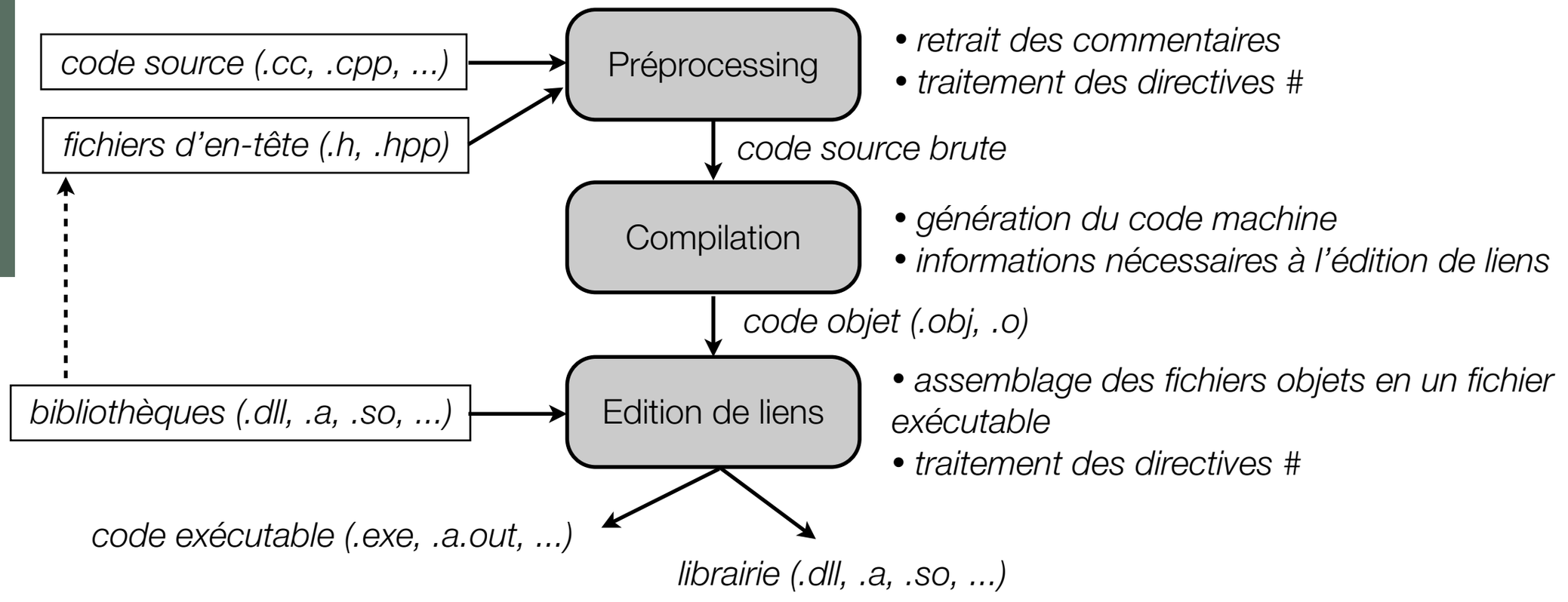
# UE Programmation C++

Master Sciences pour l'Ingénieur



## Introduction

- **Le langage C++** est un langage compilé
- **La compilation :** série d'étapes qui transforment un code source en un code machine exécutable sur un processeur cible.
- **Langage interprété :** langage qui nécessite un logiciel pour interpréter le source lors de son exécution, contrairement au langage compilé qui est directement exécuté.



## Les flux standards



L'accès aux entrées-sorties simples en C (opérations de saisie à partir du clavier et d'affichage à l'écran) se fait en utilisant les fonctions printf et scanf. On dispose en C++ de trois variables pour réaliser les mêmes opérations:

**cin:** le flux standard d'entrée

**cout:** le flux standard de sortie

**cerr:** le flux standard d'erreur

Ces trois flux sont l'équivalent de stdin, stdout et stderr en C. Leur utilisation requiert d'inclure le fichier d'en-tête iostream.h :

```
#include <iostream.h>
```

ou, pour les compilateurs plus modernes :

```
#include <iostream>
#using namespace std;
```

# Les flux standards



La lecture/écriture sur les flux standards s'effectue en utilisant deux opérateurs propres au langage C++ :

**l'opérateur d'injection noté <<**

**l'opérateur d'extraction noté >>**

Par exemple, afficher le texte bonjour s'effectuera ainsi :

```
cout << "bonjour";
```

afficher un nombre s'effectuera également de la même manière :

```
cout << 10;
```

et afficher un ensemble de données pourra se faire ainsi :

```
cout << "voici ma donnée à afficher: " << 10 << ".";
```

## Les flux standards

Pour extraire une donné du flux standard d'entrée (le clavier par défaut), on procède à l'aide de l'opérateur d'extraction :

```
int n;
double x;
cin >> x >> n;
```

L'exemple suivant illustre l'emploi des deux flux :

```
#include <iostream>
#using namespace std;
void main() {
  char nom[10];
  int age;
  cout << "Quels sont votre nom et votre âge: ";
  cin >> nom >> age;
  cout << "Vous vous appelez " << nom << " et vous avez " << age << "ans" << endl;

}
```

## Déclaration des variables



En C++ la déclaration d'une variable peut se faire à tout endroit du code. Il n'en demeure pas moins qu'une variable ne peut être utilisée qu'après avoir été déclarée ! Ainsi l'exemple suivant est admis en C++ alors qu'il ne l'est pas en C :

```
void main () {
   instruction1;
   instruction2;
   int a=10;
}
```

Un intérêt majeur est que cela permet de déclarer une variable au moment où on peut l'initialiser et pas avant. Prenons l'exemple suivant de l'allocation dynamique d'un tableau d'entiers en C :

```
void main () {
   int *tab;
   instruction1;
   instruction2;
   ...;
   tab = (int*) malloc ( 10*sizeof(int) );
   tab[0] = 0;
   ...;
}
```

## Déclaration des variables



Sa version en C++ permettra d'éviter l'usage de la variable tab avant qu'elle n'ait été initialisée, ce qui conduirait à des erreurs d'exécution :

```
void main () {
    instruction1;
    instruction2;
    ...;
    int *tab = (int*) malloc ( 10*sizeof(int) );
    tab[0] = 0;
    ...;
}
```

# Portée des variables



La zone du code (en dehors d'une fonction, dans une fonction, dans un bloc d'instructions) conditionne la portée d'une variable:

- en dehors de toute fonction : il s'agit d'une variable globale dont la portée s'étend à l'ensemble du fichier source
- dans une fonction : il s'agit d'une variable locale dont la portée est restreinte à la fonction à partir de sa ligne de déclaration
- dans un bloc d'instruction: la portée de la variable est restreint au bloc en question.
- en dehors du fichier source: il s'agit d'une variable externe déclarée dans un autre fichier (Compilation séparée)

# Portée des variables: exemple



```
int var_globale = 10;                    // il s'agit d'une variable globale

void main () {
   int var_locale;                       // il s'agit d'une variable locale à la fonction
   instruction1;
   instruction2;
   ...;
   float var_locale2;                    // variable locale dont la portée commence ici
   int i;
   for ( i=0; i<10; i++) {
      int x=i*i;                         // portée de x est limitée au bloc d'instruction;
      cout << i << " au carré = " << x << "\n";
   }
   cout << "dernier résultat: x = " << x << "\n";    // cette ligne génère une erreur à la c
}
```

## Portée des variables: exemple



Il est également possible de déclarer la variable d'itération d'une boucle dans la définition de la boucle :

```
(...)
for ( int i=0; i<10; i++) {                    // la portée de i est le corps de la boucle
    int x;
    x = i^2;

    cout << i << " au carré = " << x << "\n";
}
cout << "dernière valeur de i = " << i << "\n";   // cette ligne génère une erreur à la
}
```

## Classe de stockage des variables

On distingue plusieurs classes de stockage des variables parmi lesquelles :

- la classe **auto** qui est la classe par défaut des variables
- la classe **static** qui permet de conserver la valeur d'une variable locale lorsque l'on quitte la fonction où elle a été déclarée
- la classe **register** qui stocke la valeur d'une variable dans un registre du processeur
- la classe **extern** déjà introduite plus haut

Dans l'exemple suivant on retrouve la valeur de la variable var lors du deuxième appel à la fonction telle qu'elle était lors du premier appel :

```
#include <iostream>
using namespace std;
void fonction() {
    static int i = 0;
    i++;
    cout << "i=" << i << "\n";
}
void main() {
  fonction();
  fonction(); }
```

affichera:
i=10
i=2

## Le type booléen

Le type 'booléen n'existe pas dans le langage C mais existe en C++. Dès lors il devient inutile, et en fait déconseillé, d'utiliser des entiers pour des booléens et vice-versa. Par exemple on n'écrira pas :

```
int var;
...
if ( !var )
```

mais :

```
int var;
...;
if ( var == 0 ) ...;
```

en revanche le code suivant sera correcte :

```
bool flag=true;
...;
if ( flag ) ...;
```

## Fonctions: paramètres formels



On désigne ainsi les variables locales déclarées en paramètre d'une fonction. Par exemple la fonction suivante élève un nombre x à la puissance n :

```
double puissance( double x, int n) {
    for (int i=1; i<n; i++)
        x = x*x;
    return x;
}
```

Les variables x et n sont locales à la fonction et appelées paramètres formels de la fonction. Ces paramètres peuvent être initialisés par défaut lors de l'appel à la fonction à condition de déclarer ainsi le prototype de la fonction (pour une initialisation par défaut à 2 du paramètre n) :

```
double puissance( double x, int n=2 ) {
    for (int i=1; i<n; i++)
        x = x*x;
    return x;
}
```

## Fonctions: paramètres formels



L'initialisation par défaut des paramètres permet d'utiliser la fonction sans spécifier les paramètres en question. Ainsi, l'appel à la fonction puissance en oubliant volontairement le deuxième argument transformera cette dernière en fonction d'élévation au carré :

```
int main() {
    ...;
    cout << "10 élevé au carré = " << puissance( 10 ) << "\n";
}
```

appel équivalent à :

```
int main() {
    ...;
    cout << "10 élevé au carré = " << puissance( 10, 2 ) << "\n";
}
```

## Surcharge des fonctions



Supposons que l'on cherche à élever à une puissance quelconque des nombres pouvant être entiers ou flottants. En C il faudra écrire deux fonctions :

```
int puissance_int( int x, int n) {
    for (int i=1; i<n; i++)
        x = x*x;
    return x:
}

double puissance_dbl( double x, int n) {
    for (int i=1; i<n; i++)
        x = x*x;
    return x:
}
```

## Surcharge des fonctions

En C++ la surcharge du nom des fonctions permet de donner le même nom à ces deux fonctions :

```
int puissance( int x, int n) {
    for (int i=1; i<n; i++)
        x = x*x;
    return x:
}

double puissance( double x, int n) {
    for (int i=1; i<n; i++)
        x = x*x;
    return x:
}
```

Le compilateur saura quelle fonction appeler en observant le type des paramètres effectifs, c'est à dire des paramètres utilisés lors de l'appel à la fonction. Par exemple, dans :

```
double var;
var = puissance( var, 3 );
```

le compilateur utilise la deuxième fonction car le type du premier paramètre effectif utilisé (la variable var) est le type double.

## Surcharge des fonctions: les règles



Ce qui permet au compilateur de choisir la bonne fonction, c'est le type des paramètres formels de la fonction. La liste des types des paramètres d'une fonction s'appelle la signature de la fonction. Ainsi, la signature de la fonction puissance s'appliquant à des valeurs double est :

(double, int)

Lors de l'appel d'une fonction disposant de plusieurs versions (on dira d'une fonction surchargée), c'est la signature de la fonction comparée aux types des arguments effectifs qui décide de la fonction réellement appelée.

Attention au fait que le type retourné par la fonction ne fait pas partie de la signature de la fonction. Il est donc impossible de donner le même nom à deux fonctions dont la seule différence serait le type retourné par les deux fonctions.

## Le type référence: déclaration et initialisation



Pour chaque type de variable que l'on peut déclarer en langage C, il est possible de définir également un pointeur sur une variable de ce type. Le C++ introduit une notion supplémentaire qui est le Le type référence. Il permet de manipuler les adresses des variables en gardant la syntaxe habituelle de manipulation des variables. Une référence est en quelque sorte un pointeur géré de manière interne par le compilateur.

Les deux instructions suivantes :

```
int i;
int &ri = i;
```

déclarent une variable entière i et une référence ri sur cette variable entière. Toute opération effectuée sur ri agit sur la variable référencée, à savoir i.

```
ri = 2;
cout << "ri = " << ri << " et i = " << i << "\n";
```

affiche : ri = 2 et i = 2

l'action sur la référence de i a modifié la variable référencée, il est donc obligatoire d'initialiser une référence lors de sa déclaration.

## Le type référence: déclaration et initialisation



Ainsi:

```
ri = 2;
cout << "ri = " << ri << " et i = " << i << "\n";
```

affiche :

```
ri = 2 et i = 2
```

L'action sur la référence de i a modifié la variable référencée il est donc obligatoire d'initialiser une référence lors de sa déclaration.

```
int i;
int &ri;      // erreur générée à la compilation car la référence n'est pas initialisée
ri = i;
```

# Références en argument de fonction



Le type référence permet la transmission de l'adresse des paramètres effectifs à une fonction tout en conservant la syntaxe du passage par valeur.

Autrement dit, un argument passé par référence présente les mêmes avantages que le passage d'un pointeur, mais la syntaxe est identique à celle d'un passage par valeur.

L'exemple suivant montre trois fonctions avec en argument une structure passé par valeur, par adresse et par référence...

## Références en argument de fonction

```
struct complex {
  double r;
  double i;
};

void afficher1 ( complex cv ) {                    // les éléments de la structure sont intégralement
passés en argument
cout << "c = " << cv.r << " + i*" << cv.i << "\n";

}

void afficher2 ( complex *cp ) {                   // seule l'adresse de la structure est passée en
argument
cout << "c = " << cp->r << " + i*" << cp->i << "\n";

}

void afficher3 ( const complex &cr ) {             // l'adresse est transmise sous la forme d'une
référence
cout << "c = " << cr.r << " + i*" << cr.i << "\n";

}

void main() {
  complex c;
  c.r = 2.;
  c.i = 2.;
  afficher1( c );  afficher2( &c );  afficher3( c );
}
```

## Allocation dynamique de mémoire



L'allocation mémoire en C++ s'effectue à l'aide des opérateurs **new** et **delete**
L'allocation dynamique de mémoire en C nécessite l'emploi des fonctions malloc pour l'allocation et free pour la libération. Ces deux fonctions peuvent toujours être utilisées en C++ mais on leur préférera les deux opérateurs du langage new et delete:

pour l'allocation de mémoire :

```
int n = 10;
char *p = new char [n];          // allocation d'un tableau de n=10 octets de mémoire
int *var = new int;              // allocation d'une variable entière
```

pour la libération de mémoire :

```
delete [] p;                     // restitution des 10 octets de mémoire du tableau
delete var;                      // restitution de la place mémoire allouée à la variable
```

# Allocation dynamique de mémoire



Selon que l'on alloue un tableau (quel que soit le type des données du tableau) ou une variable simple, on emploie les crochets avec les opérateurs **new** et **delete** ou pas. Le nom respect de cette syntaxe peut provoquer un comportement indéterminé du programme. L'intérêt d'utiliser **new** et **delete** en lieu et place des fonctions malloc et free du C s'explique par le comportement de ces opérateurs lors de l'allocation dynamique d'objets: les constructeurs et destructeurs de ces objets sont en effet automatiquement appelés, une fois lorsqu'il s'agit d'allouer des objets simples, n fois lorsqu'il s'agit d'allouer des tableaux de n objets.

Deuxième argument: les opérateurs **new** et **delete** sont d'un emploi plus simple comme le montre la comparaison suivante :

```
double *p = (double *) malloc ( 50 * sizeof(double) );        // en C
double *p = new double [50];                                   // la même chose en C++
```